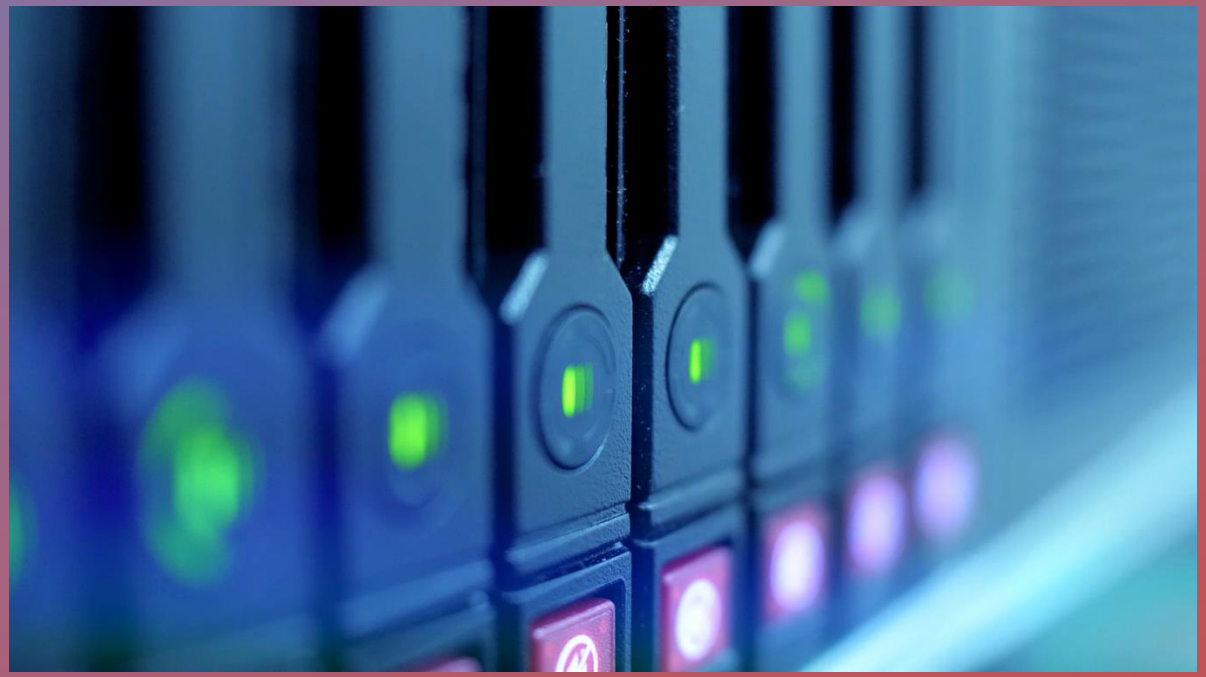
Distributed Machine Learning 2: FSDP and Inference

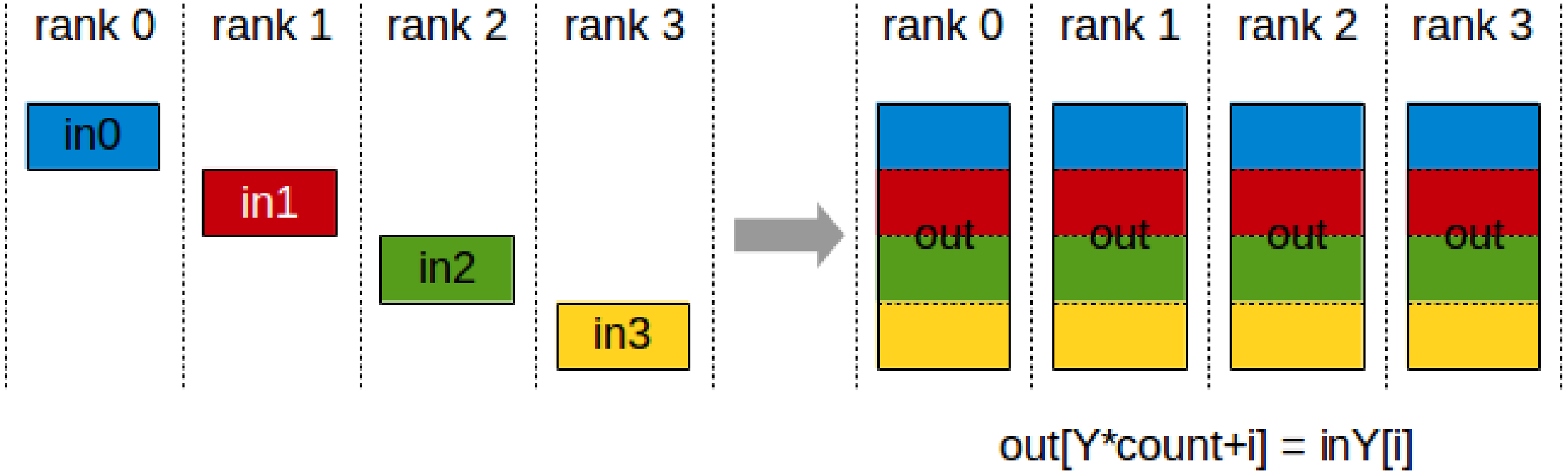


CS4787 Lecture 21 — Fall 2025

Basic patterns of collective communication

All-Gather

- Each machine has an equal-sized chunk of a desired result; sends data to many machines.

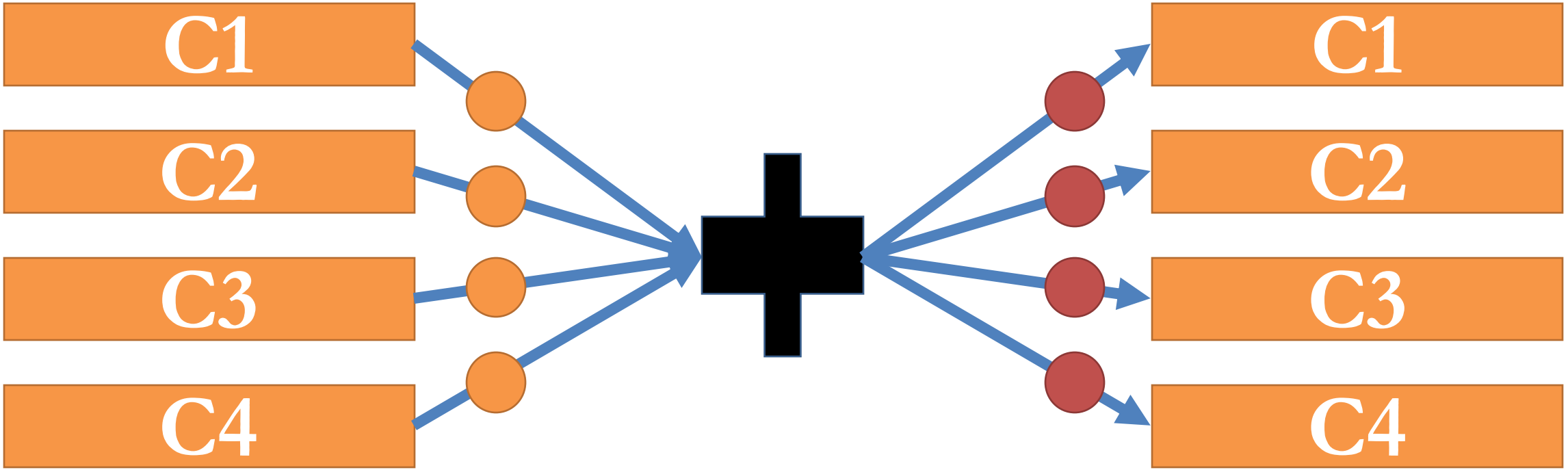


https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/usage/collectives.html#allgather

Recall: Basic patterns of collective communication

All-Reduce

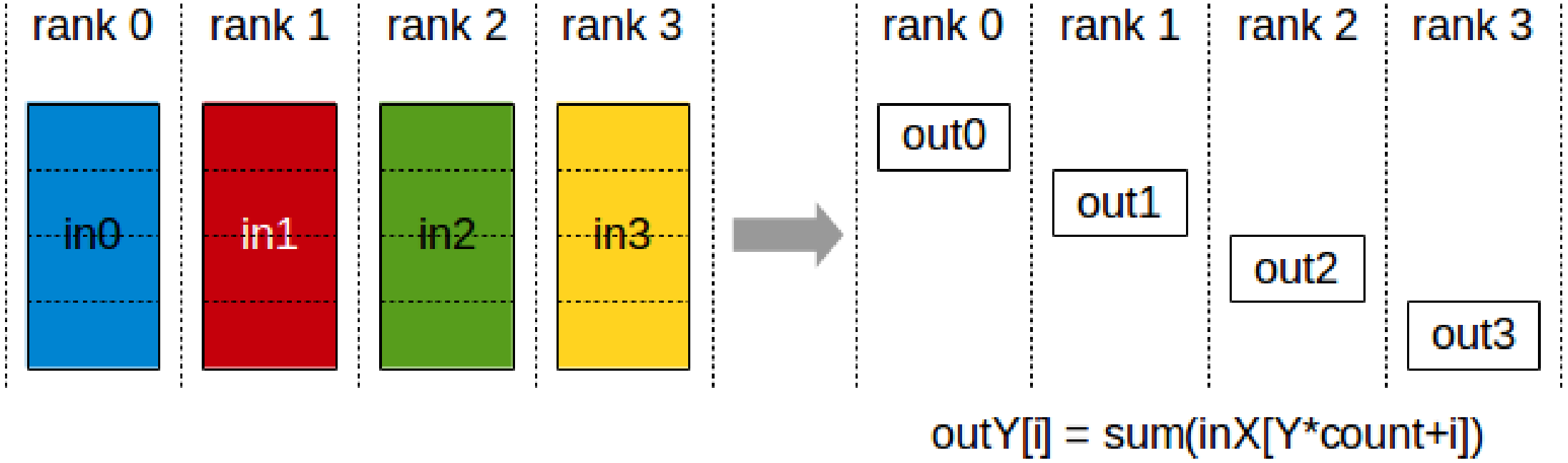
- Compute some reduction (usually a sum) of data on multiple machines and materialize the result on all those machines.



Basic patterns of collective communication

Reduce-Scatter

- Compute some reduction of data on **M** machines and materialize **1/M** of the result on each machine (sharding the result).



Recall: Data Parallel Training

- If there are M worker machines such that $B = M \cdot B'$, then

$$w_{t+1} = w_t - \alpha_t \cdot \frac{1}{M} \sum_{m=1}^{M} \frac{1}{B'} \sum_{b=1}^{B'} \nabla f_{i_{m,b,t}}(w_t).$$

- Parallelize over the minibatch, by splitting the training examples among the workers.

- We discussed two types:
  - SGD with All-Reduce
  - Parameter server architecture

Another Type of Data Parallel Learning: Federated learning

- Sometimes, **your data is inherently distributed**
  - For example, data gathered on people's mobile phones
  - For example, data measured by internet-of-things devices

- Rather than centralizing the data, may want to learn on the distributed devices themselves
  - E.g. to preserve the privacy of users

- This is called **federated learning**
  - **Lots of interest from industry right now**

Model Parallelism

- Broad concept of **partitioning the layers** of a neural network among different worker machines. (Can also partition within a layer.)

- Each worker is responsible for a subset of the parameters.

- Forward and backward signals running through the neural network during backpropagation now also run across the computer network between the different parallel machines.
  - Particularly useful if the parameters won't fit in memory on a single machine.
  - This is very important when we move to specialized machine learning accelerator hardware, where we're running on chips that typically have limited memory and communication bandwidth.

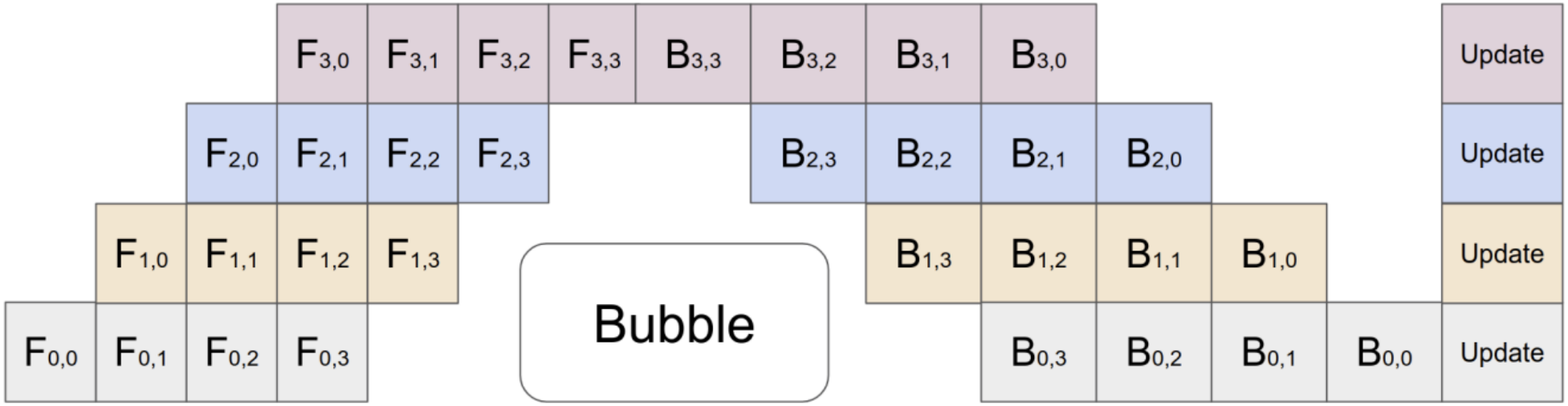
Two broad classes of model parallelism

- Distributing across layers
  - E.g. if a model has 80 transformer blocks and we're parallelizing across 8 machines, each machine will get 10 of the transformer blocks

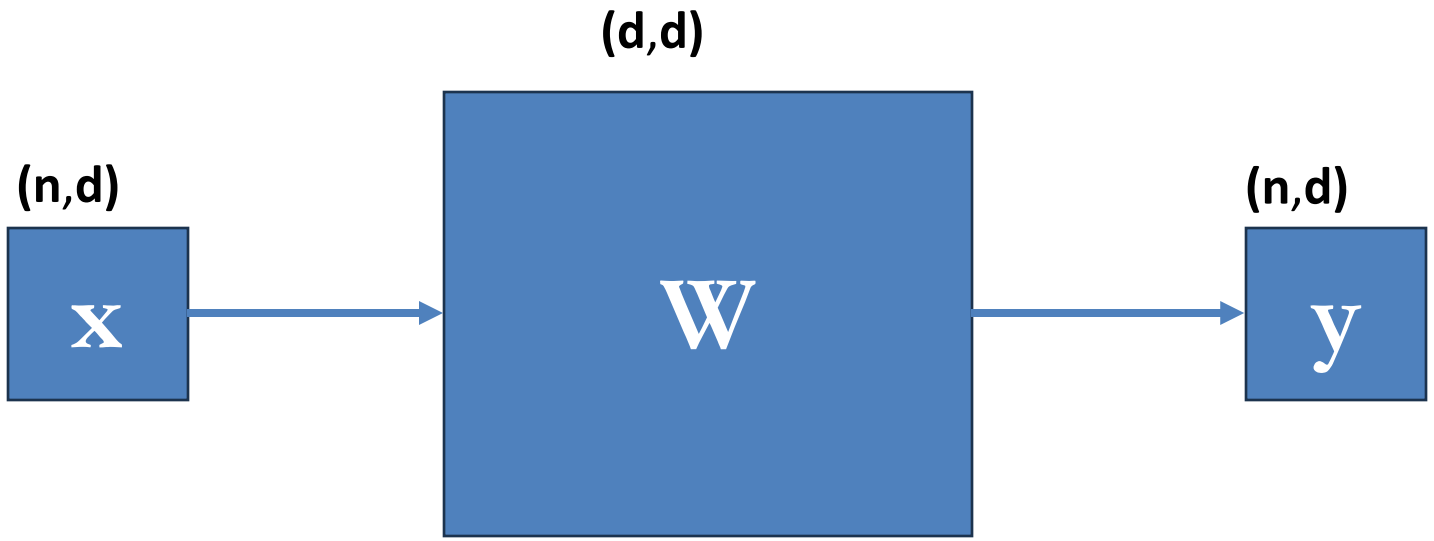
- Distributing within layers
  - Each linear layer of the model is sharded across the machines
  - Can do this on the **input** side or on the **output** side

Distributing-Across-Layers: Pipeline Parallelism

- A variant of model parallelism that tries to improve throughput by overlapping minibatch computation.
  - From *"GPipe: Easy Scaling with Micro-Batch Pipeline Parallelism"*

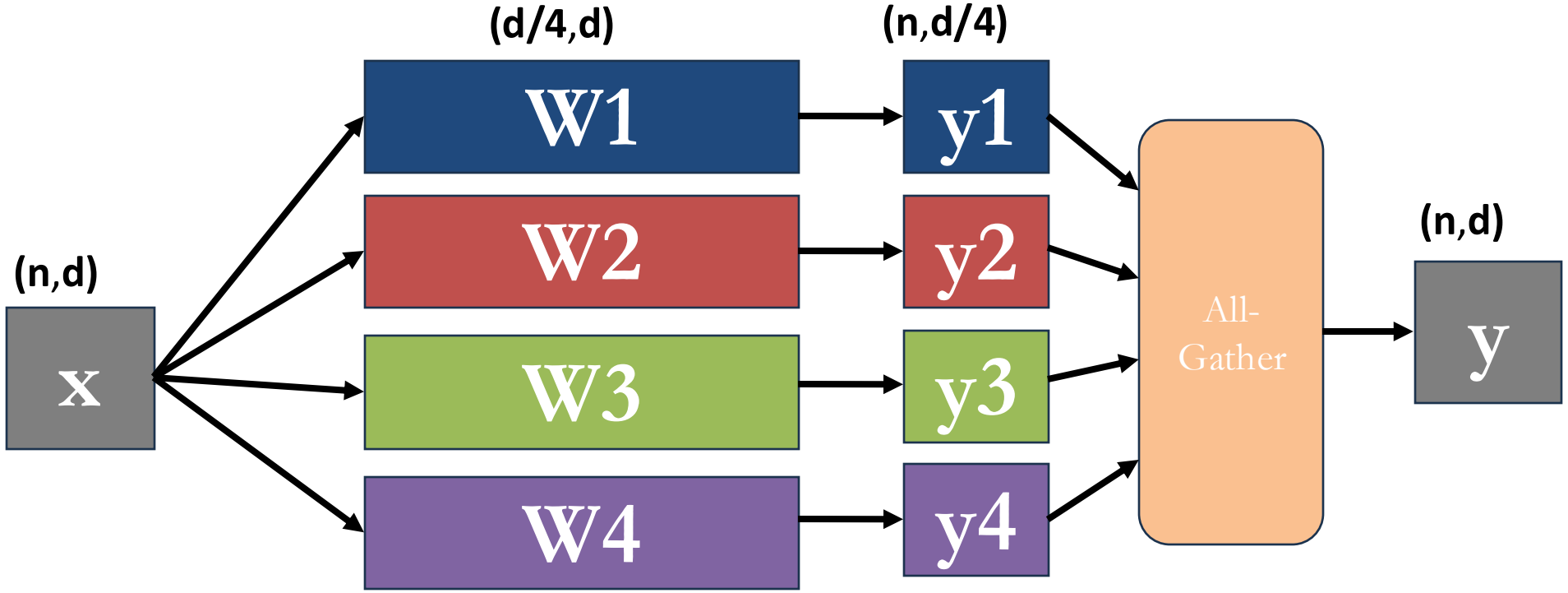


Distributing within Layers: Naïve Tensor Parallelism



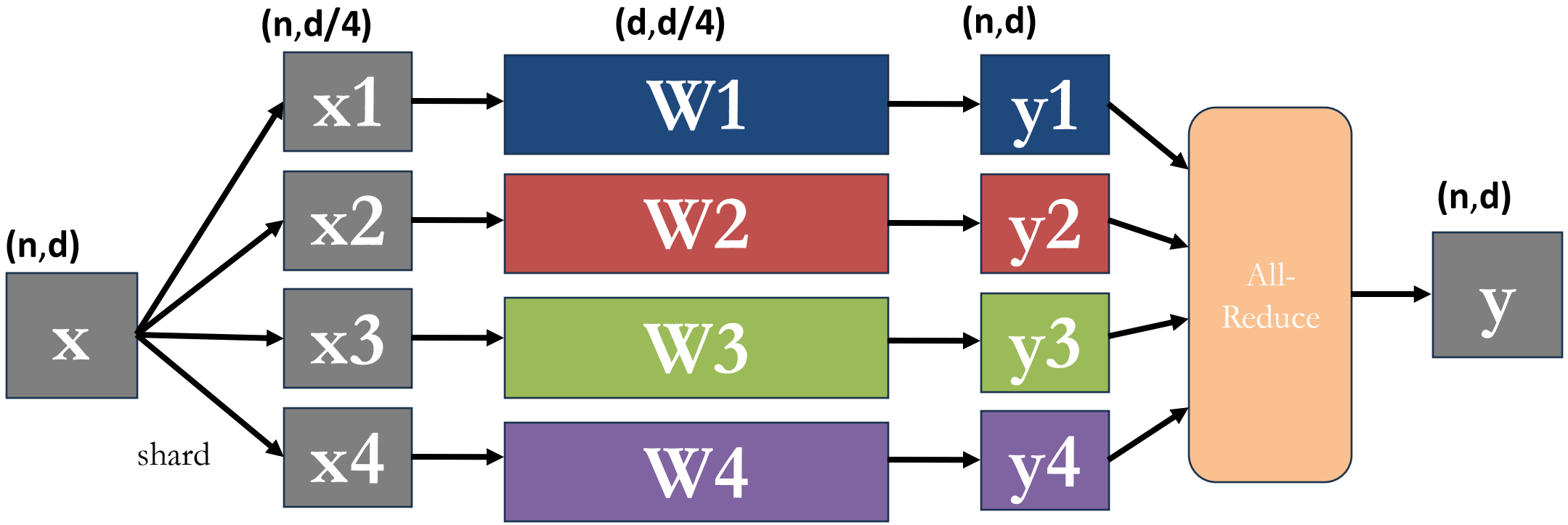
Consider a simple linear layer.

Naïve Tensor Parallelism on Output via All-Gather



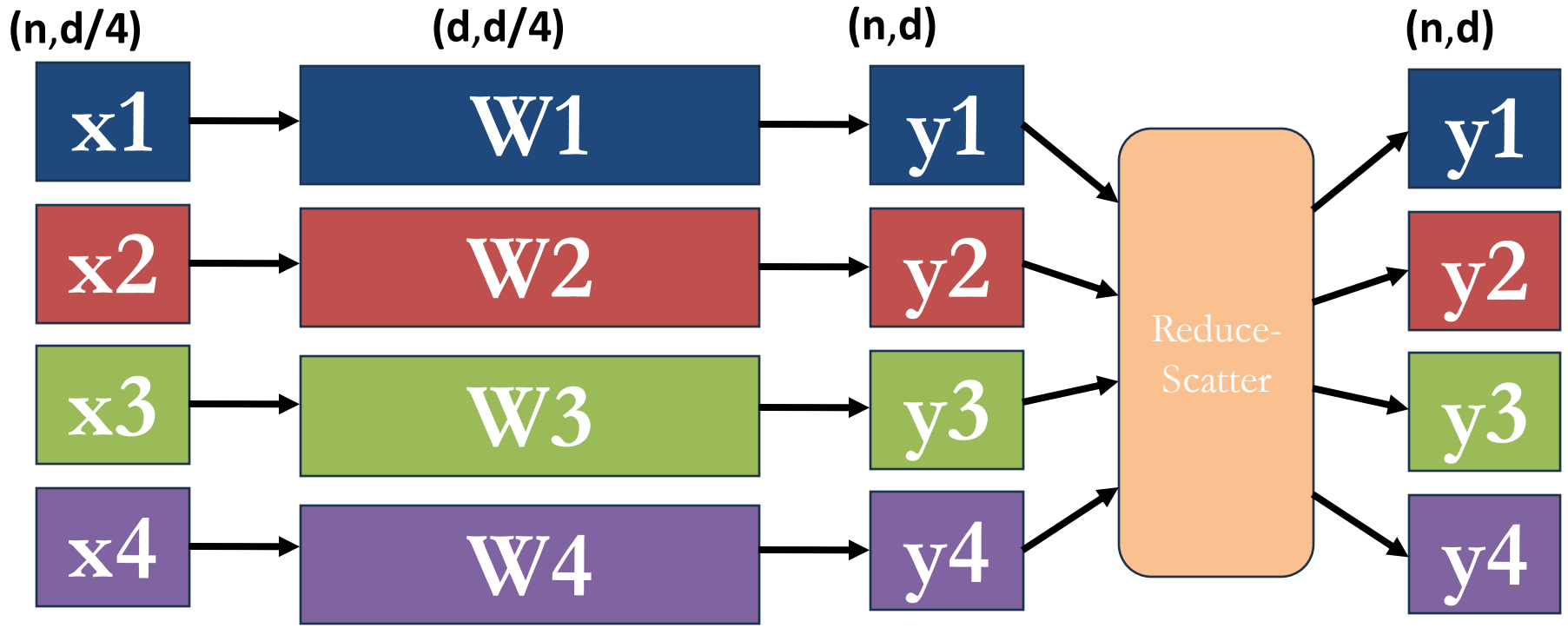
Suppose the input activations x are replicated on all workers (gray). We can partition the weights as follows.

Naïve Tensor Parallelism on Input via All-Reduce



Suppose the input activations x are replicated on all workers (gray). We can partition the weights as follows.

Naïve Tensor Parallelism on Input via Reduce-Scatter



Suppose the input activations x are sharded among workers. We can partition the weights as follows.

Key insight: we aren't forced to use the same strategy to parallelize all layers.

Can distribute some weight matrices along the input and others along the output!

Transformer MLP Block

$$y = ((xW_{\text{up}}^T) \odot \text{SiLU}(xW_{\text{gate}}^T))W_{\text{down}}^T$$

```
class LlamaMLP(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.config = config
        self.hidden_size = config.hidden_size
        self.intermediate_size = config.intermediate_size
        self.gate_proj = nn.Linear(self.hidden_size, self.intermediate_size, bias=config.mlp_bias)
        self.up_proj = nn.Linear(self.hidden_size, self.intermediate_size, bias=config.mlp_bias)
        self.down_proj = nn.Linear(self.intermediate_size, self.hidden_size, bias=config.mlp_bias)
        self.act_fn = ACT2FN[config.hidden_act]

    def forward(self, x):
        down_proj = self.down_proj(self.act_fn(self.gate_proj(x)) * self.up_proj(x))
        return down_proj
```

Can you find a way to distribute this with only 1 all-reduce?

Suppose the inputs/outputs are replicated on all machines.

Transformer Attention Block

```
def forward(self,hidden_states,positions,attention_mask):
    input_shape = hidden_states.shape[:-1]
    hidden_shape = (*input_shape, -1, self.head_dim)
    query_states = self.q_proj(hidden_states).view(hidden_shape).transpose(1, 2)
    key_states = self.k_proj(hidden_states).view(hidden_shape).transpose(1, 2)
    value_states = self.v_proj(hidden_states).view(hidden_shape).transpose(1, 2)
    query_states, key_states = self.apply_rotary_pos_emb(query_states,key_states,positions)
    attn_output = self.multi_head_attention(query_states,key_states,value_states,attention_mask)
    attn_output = attn_output.reshape(*input_shape, -1).contiguous()
    attn_output = self.o_proj(attn_output)
    return attn_output
```

Can you find a way to distribute this with only 1 all-reduce?

Suppose the inputs/outputs are replicated on all machines.

Transformer Tensor Parallelism

- Conclusion: we only need **2 all-reduce operations** to compute a single transformer block
  - Even though that block has 7 linear layers!

- Also naturally results in the **KV cache** being distributed as well
  - Helps support long sequences

Transformer Tensor Parallelism: Scaling

- Forward pass communication scales with
  - Number of transformer blocks (depth)
  - Hidden dimension (width)
  - Sequence length
  - Batch size

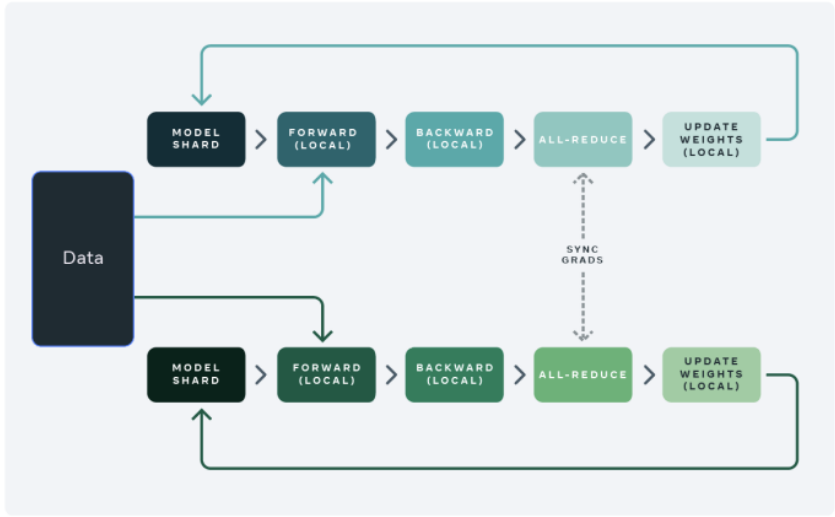
$$2 \cdot B \cdot n \cdot d \cdot L$$

Distributing within Layers: Fully Sharded Data Parallel

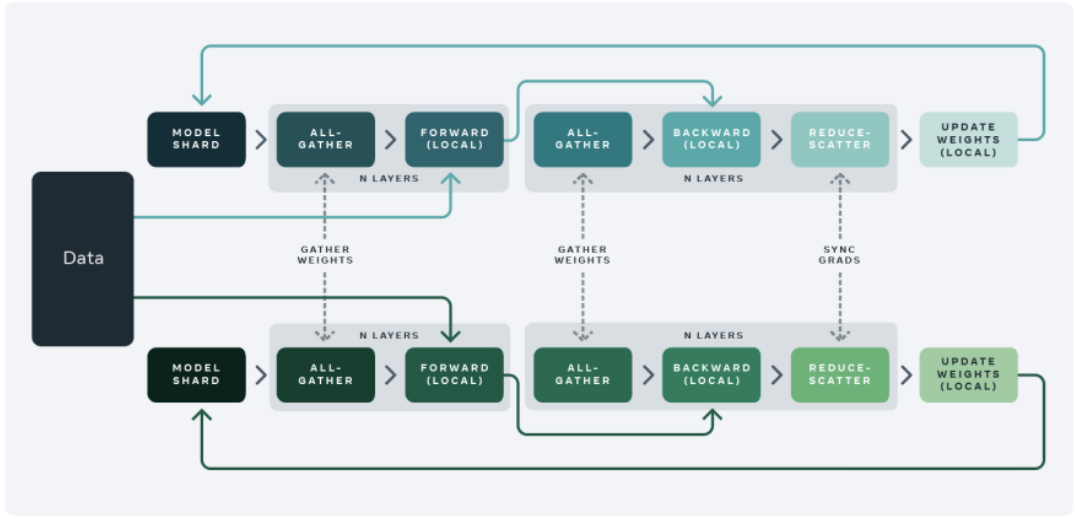
- A hybrid of data parallelism and sharded parameter server strategies.

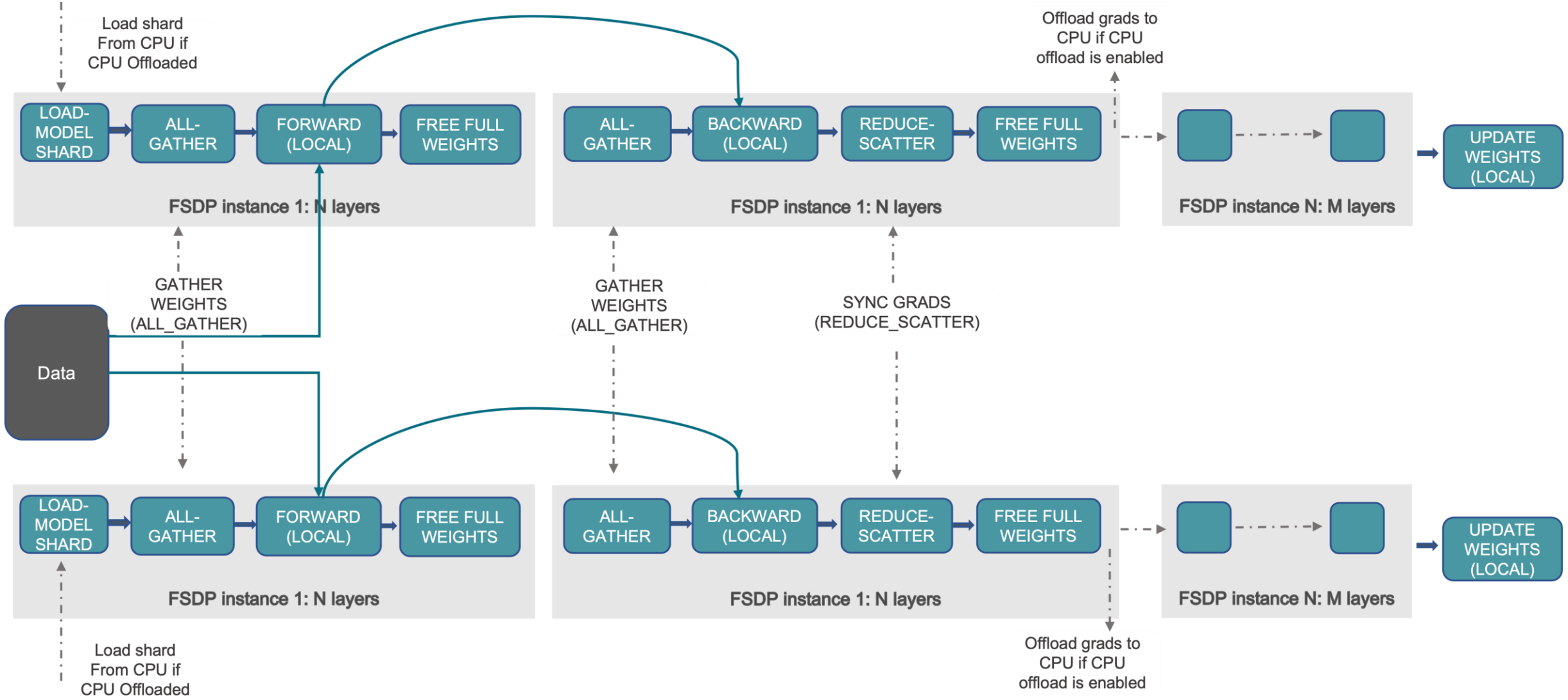
- Splits the weights for each layer among all machines, then uses an all-gather to get them whenever they're needed.

Standard data parallel training



Fully sharded data parallel training





https://pytorch.org/blog/introducing-pytorch-fully-sharded-data-parallel-api/

Transformer FSDP: Scaling

- Forward pass communication scales with
  - Total model size, since each weight is all-gathered to all machines once during the forward pass

$$L \cdot d \cdot (4d + 3d_{\text{MLP}})$$

- Better than tensor parallelism when the batch size and/or sequence length is large!

Fault-Tolerance

- Central topic in distributed computing: **what to do when a machine or network component fails?**
  - More likely to happen the more machines you're running on!

- Classic strategy: just re-run the job

- To avoid having to re-run the whole job, we periodically **checkpoint the weights/optimizer state of the model**.
  - Lets us resume from that checkpoint later in the event of failure.

Note on Activation Checkpointing

- Note that this weight checkpointing is not to be confused with **activation checkpointing** (a.k.a. gradient checkpointing).

- Activation checkpointing happens while computing the gradient
  - Main idea is to save memory for the intermediates used in backprop
  - Save only the activations going into a block
  - Recompute the other activations on the fly

- Trades off **memory for compute**
  - Effectively need to run the forward pass twice

Distributed LLM inference
Why should we care about inference?

- **Train once, infer many times**
  - Many production machine learning systems just do inference

- Often want to run inference on **low-power edge devices**
  - Such as cell phones, security cameras
  - **Limited memory** on these devices to store models

- Need to get **responses to users quickly**
  - On the web, users won't wait more than a second

Recall: Inference on neural networks

- Just need to run the forward pass of the network.
  - A bunch of matrix multiplies and non-linear units.

- Unlike backpropagation for learning, here we **do not need to keep the activations around for later processing**.
  - Lower memory requirements

- This makes inference a much simpler task than learning.
  - Although it can still be costly — it's a lot of linear algebra to do.

What performance metrics do we care about when running inference?

E.g. for a deep neural network application

Metrics for Inference

- Important metric: **throughput**
  - **How many examples** can we classify in some amount of time
- Important metric: **latency**
  - **How long** does it take to get a prediction for a single example
- Important metric: **model size**
  - **How much memory** do we need to store/transmit the model for prediction
- Important metric: **energy use**
  - **How much energy** do we use to produce each prediction
- **What are examples where we might care about each metric?**

Distributed LLM inference

- Mostly done via **tensor parallelism**
- Key thing to remember: for **decode** latency we are memory-bound, since we need to read all the model weights into memory to make each prediction
- For **prefill** we still want to be compute bound
  - Looks like training computationally

Distributed LLM inference

- Because **decode is memory bound**, we leverage multiple devices to get higher memory bandwidth
  - 8x the GPUs means 8x the memory bandwidth
  - …means ideally an 8x lower latency to infer one token

- But we still want to use the compute, so to be efficient, **batch multiple inference requests together**
  - This both lets us use otherwise underused FLOPs
  - And lets us do more efficient matrix-matrix multiplies instead of matrix-vector

Distributed LLM inference: batching

- As a result, there's this fundamental property of LLM inference: **you can be more efficient at scale!**
  - Inference systems have a lower cost per token once they reach a certain threshold of queries
  - …so they can always operate at an "ideal" batch size
- This gives major AI inference companies an advantage over running your own system locally!

Why set up your own distributed inference?

- **Privacy**
  - You can keep your queries secure
  - You can keep your model weights secure
- **Cost** (maybe)
  - Just because cloud inference providers can do it more efficiently, doesn't mean the market rate will necessarily be cheaper
- **Latency**
  - For some applications, the time to go over the internet to a cloud provider might be too large (e.g. a self-driving car, or a trading system)

Conclusion and Summary

- Distributed computing is a powerful tool for scaling machine learning: both training and inference
- If you use methods specialized to your own ML model (and design the ML model with these methods in mind!) you can make efficient use of distributed hardware.
- New **Paper Reading** out tonight: MegatronLM